Una presentación del algoritmo

Primes in P

Agrawal, Kayal, Saxena

Cristian Ruz R.
PUC 2002

Introducción

PRIMALIDAD

Entrada: Un entero p, $p > 1$.

Salida: `PRIMO`, si p es un número primo.
`COMPUESTO`, si p es un número compuesto.

A lo largo de la historia se han desarrollado una larga lista de algoritmos para resolver este problema, pero muchos son de tiempo exponencial, o probabilísticos.

- La criba de Eratóstenes (Eratosthenes' Sieve).
  Tiempo $O(n)$, donde n es el número de entrada, por lo tanto es exponencial en el tamaño de la entrada. ($\log n$)

- *Pequeño Teorema de Fermat.* $a, p \in \mathbb{Z}$

$$\forall p \text{ primo}, p \text{ no divide a } a \Rightarrow a^{p-1} \equiv 1 \pmod{p}$$

Lamentablemente el converso no se cumple (números de Carmichael).

Se han desarrollado algoritmos probabilísticos de test de primalidad a partir de esto. Al menos PRIMALIDAD $\in NP$. Pero se necesita un algoritmo determinístico de tiempo polinomial para demostrar que está en P.

- Wilson (1770). p es primo $\Leftrightarrow (p-1)!+1 \equiv 0 \pmod{p}$.

- Test de Miller (1976). Polinomial y determinístico ... siempre que se cumpla la *Hipótesis Extendida de Riemann* .

- Rabin-Miller (1980). Solovay y Strassen (1977). Polinomial, pero probabilístico.

- Adleman, Pomerance, Rumely (1983). Determinístico, pero no polinomial. $O((\log n)^{O(\log\log\log n)})$

- Goldwasser y Killian (1986). Polinomial y probabilístico, modificado por Atkin y Morain (1993). Basado en curvas elípticas.

El Algoritmo AKS

Solo se sabía que PRIMALIDAD $\in NP \cap$ co-NP

Todo era así hasta el 6 de Agosto de 2002.

El algoritmo de Agrawal, Kayal y Saxena determina en tiempo polinomial $\tilde{O}((\log n)^{12})$ si un número es primo o no.

$\tilde{O}(t(n)) \equiv O((t(n)poly(\log t(n)))$

Basado en la siguiente identidad.

Sean a, p coprimos ($\gcd(a,p) = 1$))

$$p \text{ es primo } \Leftrightarrow (x-a)^p \equiv (x^p - a) \pmod{p}$$

$$p \text{ es primo } \Leftrightarrow (x-a)^p \equiv (x^p - a) \pmod{p}$$

Si p es primo $\Rightarrow \forall i, 0 < i < p$, los coeficientes del binomio $(x-a)^p$ son $(-1)^i \binom{p}{i} a^{p-i} = p \cdot k \equiv 0 \pmod{p}$. Luego,

$$(x-a)^p \equiv x^p - a^p \equiv x^p - a \pmod{p}, \text{ (Fermat)}$$

Si p es compuesto $\Rightarrow$ sea q un factor primo de $p \Rightarrow q^k | p$. $(p = cq^k)$

q^k *no divide* a $\binom{p}{q}$, y a^{p-q}, q^k son coprimos $\Rightarrow$ coeficiente $x^q \not\equiv 0$.

Entonces el algoritmo sería calcular $(x-a)^p \pmod{p}$. Calcular $p-1$ coeficientes ya es $\Omega(p)$.

La idea es aplicar módulo $(x^r - 1)$. Se obtiene un polinomio de grado $r-1$. "Nos fijamos solo en los últimos r términos". De esta forma, la evaluación se puede hacer en $O(r \log^2 p)$.

Hay que elegir un r adecuado. Suficientemente grande para que ningún compuesto escape la prueba, y no tan grande que haga el algoritmo exponencial.

Se postula que : existe un r adecuado, primo, en un rango $O(\log^6 p)$, y tal que $r-1$ tiene un factor primo q, $q > r^{\frac{1}{2}+\delta}$, para algun $\delta > 0$, constante.

Además no es necesario verificar para todos los a. Solo para $O(\sqrt{r} \log p)$ valores de a.

El Algoritmo

Input: $n \in \mathbb{N} > 1$.

`if` (n es de la forma $a^b, b > 1$) `output COMPUESTO`;
$r = 2$;
`while` $(r < n)\{$
    `if`$(\gcd(n, r) \neq 1)$ `output COMPUESTO`;
    `if`(r es primo)
        $q =$ el mayor factor primo de $r - 1$;
        `if`$(q \geq 4\sqrt{r}\log n) \wedge (n^{\frac{r-1}{q}} \not\equiv 1 \pmod r)$
            `break`;
    $r = r + 1$;
$\}$
`for`($a = 1$ `to` $2\sqrt{r}\log n$)
    `if`$((x - a)^n \not\equiv (x^n - a) \pmod{x^r - 1, n})$ `output COMPUESTO`;
`output PRIMO`;

Correctitud del algoritmo

La correctitud del algoritmo se establece en base a una serie de lemas y demostraciones matemáticas de teoría de números.

El *loop* de las líneas 3-10 busca el r adecuado.

Lema 4.2. Para el número n existe un primo r en el intervalo $[c_1(\log n)^6, c_2(\log n)^6]$, tal que $r-1$ tiene un factor primo $q \geq 4\sqrt{r}\log n$, y $q | o_r(n)$. ($o_r(n)$ es el orden de n módulo r).

Se demuestra que existe al menos un primo en dicho intervalo $O((\log n)^6)$ que cumple esas condiciones, y por lo tanto el *loop* termina.

Lema 4.3. Si n es primo, entonces el algoritmo retorna `PRIMO`.

Dado que n es primo, entonces el ciclo **`while`** no podría retornar `COMPUESTO`, ya que en la línea 4, $\text{gcd}(n, r) = 1, \forall r \leq c_2(\log n)^6$.

Otro resultado de teoría de números dice que para un polinomio $f(x)$ con coeficientes enteros, y p primo, $f(x)^p \equiv f(x^p) \pmod{p}$. Por lo tanto la línea 12 tampoco puede retornar `COMPUESTO`.

Lema 4.7. Si n es compuesto, el algoritmo retorna `COMPUESTO`.

Suponiendo que el algoritmo retorna `PRIMO`. Luego la condición dentro del ciclo `for` se ha cumplido para todo a, $1 \leq a \leq 2\sqrt{r} \log n$.

$$((x-a)^n \not\equiv (x^n - a) \pmod{x^r - 1, p})$$

Se demuestra que la única forma que esta prueba sea burlada es que $n = p^k$, para $k \geq 1$. Para esto, todos los números de esta forma se descartan en el paso 1. La opción es que $n = p$, pero p es primo. Contradicción.

Análisis de complejidad

Paso 1. Determinar si un número n es de la forma $a^b, b > 1$, se puede realizar en tiempo $O(\log^3 n)$. La idea es recorrer los números $p \leq \log n$, y en cada caso chequear el signo de $n - x^k$ para algún entero x, mediante aproximaciones.

Pasos 3-10. El ciclo `while` toma tiempo $O(\log^6 n)$, debido a que el r que se busca está en ese rango.

Paso 4. El cálculo de $\gcd(r, n)$ toma $poly(\log\log r) < poly(\log\log n)$.

Pasos 5 y 6. Se pueden implementar en $O(\sqrt{r} poly(\log\log n)$.

Paso 7. Puede tomar, a lo más $poly(\log\log n)$.

Total del ciclo `while`.

$$
\begin{aligned}
&= O((\log^6 n)\cdot(poly(\log\log n) + poly(\log\log n) + \sqrt{r}poly(\log\log n)) \\
&= O((\log^6 n)\cdot\sqrt{r}poly(\log\log n)) \\
&= O((\log^6 n)\cdot(\log^3 n)poly(\log\log n)) \\
&= O((\log^9 n)poly(\log\log n)) \\
&= O((\log^9 n)poly(\log\log^9 n) \\
&= \tilde{O}(\log^9 n)
\end{aligned}
$$

Paso 11. El ciclo `for` se ejecuta $O(\sqrt{r}\log n) = O(\log^4 n)$ veces.

Paso 12. La evaluación de la identidad, usando cuadratura repetida para la exponenciación modular, y FFT para la multiplicación, se puede hacer en tiempo $\tilde{O}(\log n \cdot r \log n)$.

Por lo tanto el ciclo `for` completo toma tiempo $O(r^{\frac{3}{2}} \log^3 n) = \tilde{O}(\log^{12} n)$.

Mejoras al algoritmo

Uno de los resultados que se utilizan afirma que para muchos primos r, se cumple que el mayor divisor primo de $p-1$, $P(r-1)$ es al menos $P(r-1) > r^{\frac{2}{3}}$. La cantidad de primos para los que se cumple esto es al menos $c\frac{r}{\log r}$, lo que ayuda a acotar la búsqueda.

Sin embargo, se cree que ara muchos primos r también se cumple que $P(r-1) = \frac{r-1}{2}$. Los números que cumplen con esta propiedad se llaman *Sophie-Germain primes*. Un número k primo es *Sophie-Germain prime* si $2k+1$ también es primo.

Si se cumple una conjetura, aún no probada pero *muy* creida, se podría reducir el tiempo del `while` a $\tilde{O}(\sqrt{r}\log^2 n)$, (ahora $r = O(\log^2 n)$)) y $\tilde{O}(r^{\frac{3}{2}}\log^3 n)$ en el `for`. El algoritmo completo sería $\tilde{O}(\log^6 n)$.

Otra mejora. El ciclo `for` es ejecuta $O(\sqrt{r}\log n)$ veces, ya que se eligió un r de tal forma que el *grupo* generado por el producto de $(x-a)^n$, módulo (x^r-1,n) es de un tamaño apropiado. De hecho es de tamaño al menos $n^{2\sqrt{r}}$. (Lema 4.4).

Si se cumple la siguiente conjetura. Si r no divide a n, y

$$(x-1)^n \equiv (x^n-1) \pmod{x^r-1,n}$$

entonces, n es primo, o bien $n^2 \equiv 1 \pmod{r}$.

Se puede modificar el algoritmo para buscar r en un rango menor, $[2, 4\log n]$. Usando FFT se puede verificar esta identidad en tiempo $\tilde{O}(r\log^2 n)$, y reducir la complejidad del algoritmo a $\tilde{O}(\log^3 n)$.

Lema 3.2 Sea $P(n) =$"el mayor divisor primo de n". Existen constante $c > 0$ y n_0 tales que para todo $x \geq n_0$.

$$|\{p|p \text{ es primo }, p \leq x, P(p-1) > x^{\frac{2}{3}}\}| \geq c\frac{x}{\log x}$$

Lema 3.3 Sea $\pi(n)$ la cantidad de primos $\leq n$, entonces para $n \geq 1$:

$$\frac{n}{6\log n} \leq \pi(n) \leq \frac{8n}{\log n}$$